6. Fortgeschrittene I/O

Fortgeschrittene I/O

- File Descriptors für Standard-I/O
- Standard-Eingabe und -ausgabe umleiten, `dup()`, `dup2()`
- Pipes
- Offene Dateien und `exec()`
- I/O Multiplexing mit `select()`
- Memory Mapped Files: `mmap()`

File Descriptors für Standard-I/O

- Jeder (normale) Prozess besitzt beim Start drei Datei-Deskriptoren für
  - Standardeingabe (`STDIN`, 0)
  - Standardausgabe (`STDOUT`, 1)
  - Standardfehlerausgabe (`STDERR`, 2)
- Zugriff darauf mit `read` / `write` wie bei anderen Dateien möglich, z. B.
  - `read (STDIN_FILENO, &buf, len)`: lesen aus Terminal
  - `write (STDOUT_FILENO, &buf, len)`: schreiben auf Terminal

`dup2()`

- `dup2()` leitet einen File Descriptor auf einen anderen um:
- `dup2 (fd1, fd2);`
  - schließt die mit `fd2` verbundene Datei
  - Ein-/ Ausgabe-Anforderungen, die an `fd2` geschickt werden, landen in `fd1`

```
#include <stdio.h>
#include <fcntl.h>
int main () {
  int fd = creat ("/tmp/out.txt", S_IRUSR | S_IWUSR);
  dup2 (fd, 1);   // stdout -> /tmp/out.txt umleiten
  printf ("Hallo, Test\n");
  return 0;
};
```

dup2()

- `stderr` auf `stdout` umleiten:

```
dup2 (1, 2);
```


dup()

- `dup()` kopiert einen File Descriptor und verwendet für die Kopie den kleinsten freien fd:
- `int fd2 = dup (fd1);`
  - Ein-/ Ausgabe-Anforderungen, die an `fd2` geschickt werden, landen in `fd1`

```
#include <stdio.h>
#include <fcntl.h>
int main () {
  int fd;
  fd = creat ("/tmp/out.txt", S_IRUSR | S_IWUSR);
  close (1);              // stdout schliessen
  dup (fd); close (fd);   // stdout -> /tmp/out.txt
  printf ("Hallo, Test\n");
  return 0;
};
```


Pipes

- aus der Shell bekannt:
  - Verknüpfung von Std.-Ausgabe eines Prozesses mit Std.-Eingabe eines weiteren
  - `prog1 | prog2`
- In C-Programmen: `pipe()` und Umleiten der File Descriptors für `stdin` bzw. `stdout`
- erzeugt zwei File Descriptors für (gemeinsame) Pipe: mit dem ersten „Ende“ lesen, mit dem zweiten schreiben

Pipes

- Implementierung der Shell-Funktion `prog1 | prog2`:
  - `pipe(fds)`, `fork()`
  - im 1. Kind:
    - `dup2 (fds[1], 1)` (stdout)
    - `fds[0]`, `fds[1]` schließen, `exec()`
  - `fork()`
  - im 2. Kind:
    - `dup2 (fds[0], 0)` (stdin)
    - `fds[0]`, `fds[1]` schließen, `exec()`
  - `fds[0]`, `fds[1]` schließen, 2x `wait()`

Pipes in C: 2 Prozesse

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>

// implements:  ls -al | grep a.out

int main () {
  int pipe_fds[2];
  int pipe_read_fd, pipe_write_fd,
      pid1, pid2;

  pipe (pipe_fds);
  pipe_read_fd  = pipe_fds[0];
  pipe_write_fd = pipe_fds[1];

  // child 1
  pid1 = fork ();
  if (pid1 == 0) {
    // pipe stuff: stdout -> fds[1]
    close (pipe_read_fd);
    dup2 (pipe_write_fd, 1);
    close (pipe_write_fd);
    // run first program
    execlp ("ls", "ls", "-al", NULL);
  }

  // child 2
  pid2 = fork ();
  if (pid2 == 0) {
    // pipe stuff: stdin -> fds[0]
    close (pipe_write_fd);
    dup2 (pipe_read_fd, 0);
    close (pipe_read_fd);
    // run second program
    execlp ("grep", "grep", "a.out", NULL);
  }

  // parent
  close (pipe_read_fd);
  close (pipe_write_fd);
  wait (NULL);
  wait (NULL);
};
```


Pipes in C: 3 Prozesse

```
// ls -al | grep a.out | tr rwx RWX

#define CLOSE_ALL_PIPES \
  close (pipe1_read_fd); \
  close (pipe1_write_fd); \
  close (pipe2_read_fd); \
  close (pipe2_write_fd);

int main () {
  int pipe1_fds[2], pipe2_fds[2];
  int pipe1_read_fd, pipe1_write_fd,
    pipe2_read_fd, pipe2_write_fd,
    pid1, pid2, pid3;
  pipe (pipe1_fds); pipe (pipe2_fds);
  pipe1_read_fd  = pipe1_fds[0];
  pipe1_write_fd = pipe1_fds[1];
  pipe2_read_fd  = pipe2_fds[0];
  pipe2_write_fd = pipe2_fds[1];

  // child 1
  pid1 = fork ();
  if (pid1 == 0) {
    // pipe stuff: stdout -> fds1[1]
    dup2 (pipe1_write_fd, 1);
    CLOSE_ALL_PIPES;
    // run first program
    execlp ("ls", "ls", "-al", NULL);
  }

  // child 2
  pid2 = fork ();
  if (pid2 == 0) {
    // pipe stuff: stdin  -> fds1[0]
    // pipe stuff: stdout -> fds2[1]
    dup2 (pipe1_read_fd, 0);
    dup2 (pipe2_write_fd, 1);
    CLOSE_ALL_PIPES;
    // run second program
    execlp ("grep", "grep", "a.out", NULL);
  }

  // child 3
  pid3 = fork ();
  if (pid3 == 0) {
    // pipe stuff: stdin  -> fds2[0]
    dup2 (pipe2_read_fd, 0);
    CLOSE_ALL_PIPES;
    // run third program
    execlp ("tr", "tr", "rwx", "RWX", NULL);
  }

  // parent
  CLOSE_ALL_PIPES;
  wait (NULL); wait (NULL); wait (NULL);
};
```


Offene Dateien und `exec()`

- Geöffnete Dateien „überleben“ den Programmaufruf mit `exec()`
- Deswegen funktioniert auch das Pipelining
- Prozess kann durch Blick in `/dev/fd/` herausfinden, welche Dateien geöffnet sind

```
$ ls -l /dev/fd/
total 0
lrwx------ 1 root root 64 Apr 28 17:06 0 -> /dev/pts/0
lrwx------ 1 root root 64 Apr 28 17:06 1 -> /dev/pts/0
lrwx------ 1 root root 64 Apr 28 17:06 2 -> /dev/pts/0
lr-x------ 1 root root 64 Apr 28 17:06 3 -> /proc/11830/fd
```


Offene Dateien und `exec()`

```
// openexec1.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main () {
  int fd;
  char s[]="Test von open und exec\n";
  fd = open ("out.txt", O_CREAT | O_TRUNC | O_RDWR);   // fd == 3
  write (fd, s, sizeof(s)); lseek (fd, 0, SEEK_SET);
  printf ("openexec1: Opened file, fd = %d\n", fd);
  printf ("openexec1: exec()-ing openexec2\n");
  execl ("./openexec2", "openexec2", NULL);
}
```

```
// openexec2.c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main () {
  char buf[100];
  int fd = 3;   // magic: file 3 is open...
  if (read (fd, &buf, 100) == -1) {
    perror("openexec2"); exit(0);
  };
  printf ("openexec2: reading from file...\n");
  printf ("%s", (char*)&buf);
}
```

```
[esser@lx:tmp]$ ./openexec2
openexec2: Bad file descriptor
[esser@lx:tmp]$ ./openexec1
openexec1: Opened file, fd = 3
openexec1: exec()-ing openexec2
openexec2: reading from file...
Test von open und exec
```

I/O Multiplexing

- Szenario:
  - mehrere offene Pipes
  - wenn auf einer Pipe „neue Eingabe“ erscheint, soll sie bearbeitet werden
  - Problem: `read()` blockiert, wenn keine Daten vorhanden sind
- Lösung:
  - nicht-blockierende I/O mit `select()`

I/O Multiplexing: select()

- `select(maxfd, &fdset1, &fdset2, &fdset3, &timeout)`
  - erhält Menge(n) von File Descriptors (`fdset1`: zum Lesen, `fdset2`: zum Schreiben, `fdset3`: mit Fehlern)
  - kehrt zurück, sobald einer davon bereit zum Lesen ist (oder der Timeout erreicht wurde)
  - vor jedem Aufruf `fdset` initialisieren:

```
FD_ZERO (&fdset);
FD_SET (fd, &fdset);   // für jeden fd
```

  - nach Rückkehr aus `select()` prüfen:

```
if (FD_ISSET(fd, &fdset)) read (fd, ...);
```

  - `maxfd`: (größter `fd`)+1

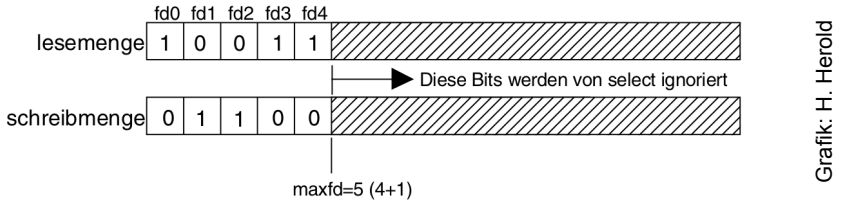
I/O Multiplexing: select()

- `select()` **schläft**, bis I/O bereit ist bzw. Timer abläuft
- Als Timeout kann auch ein Null-Pointer übergeben werden, dann wartet `select()` unendlich lange auf I/O
- Wird 0 als Timeout gesetzt, lässt sich hiermit **Polling** implementieren

I/O Multiplexing: select()

Beispiel (aus H. Herold: Linux/Unix Systemprogrammierung)

```
fd_set lesemenge, schreibmenge;
FD_ZERO(&lesemenge);
FD_ZERO(&schreibmenge);
FD_SET(0, &lesemenge);       // stdin
FD_SET(3, &lesemenge);
FD_SET(4, &lesemenge);
FD_SET(1, &schreibmenge);    // stdout
FD_SET(2, &schreibmenge);    // stderr
select(5, &lesemenge, &schreibmenge, NULL, NULL);
```



Grafik: H. Herold

Test-Szenario

```
sh1$ mkfifo /tmp/1
sh1$ mkfifo /tmp/2
sh1$ cat > /tmp/1
a
a
a
```

```
sh2$ cat > /tmp/2
b
b
b
b
b
```

```
sh3$ ./fifotest
fifo 1: a
fifo 2: b
fifo 1: a
fifo 2: b
fifo 1: a
fifo 2: b
```

```c
// fifotest.c
#include <stdio.h>
#include <fcntl.h>
#include <string.h>

int main () {
  int fd1, fd2, count;
  char buf[100];
  fd1 = open ("/tmp/1", O_RDONLY);
  fd2 = open ("/tmp/2", O_RDONLY);

  // abwechselnd aus /tmp/1, /tmp/2 lesen
  for (;;) {
    memset (buf, 0, 100);
    count = read (fd1, &buf, 100);
    if (count>0) printf ("fifo 1: %s", buf);
    memset (buf, 0, 100);
    count = read (fd2, &buf, 100);
    if (count>0) printf ("fifo 2: %s", buf);
  }
}
```

Aufrufe von `read()` blockieren

mit `select`

```
sh1$ mkfifo /tmp/1
sh1$ mkfifo /tmp/2
sh1$ cat > /tmp/1
a
a
a
```

```
sh2$ cat > /tmp/2
b
b
b
b
b
```

```
sh3$ ./fifotest
fifo 1: a
fifo 1: a
fifo 2: b
fifo 2: b
fifo 1: a
fifo 2: b
fifo 2: b
fifo 2: b
```

```c
#include <stdio.h>    #include <fcntl.h>
#include <string.h>   #include <sys/select.h>

int main () {
  int fd1, fd2, count; char buf[100];
  fd_set fds; struct timeval tv; int retval;

  fd1 = open ("/tmp/1", O_RDONLY);
  fd2 = open ("/tmp/2", O_RDONLY);
  tv.tv_sec = 1; tv.tv_usec = 0; // timeout: 1 sec

  // nach Bedarf aus /tmp/1, /tmp/2 lesen
  for (;;) {
    FD_ZERO (&fds);
    FD_SET (fd1, &fds); FD_SET (fd2, &fds);
    retval = select (fd2+1, &fds, NULL, NULL, &tv);
    if (retval == -1) perror ("select");
    if (retval == 0) continue;

    if (FD_ISSET (fd1, &fds)) {  // Daten in fd1?
      memset (buf, 0, 100);
      count = read (fd1, &buf, 100);
      if (count>0) printf ("fifo 1: %s", buf);
    }

    if (FD_ISSET (fd2, &fds)) {  // Daten in fd2?
      memset (buf, 0, 100);
      count = read (fd2, &buf, 100);
      if (count>0) printf ("fifo 2: %s", buf);
    }
  }
}
```


`select()` für Sockets

- Häufiger wird dieses Multiplexing für Netzwerk-Sockets verwendet
- Sockets werden mit `socket()` statt `open()` geöffnet, aber man kann auch diese mit `read()` und `write()` ansprechen
- Sockets haben einen Socket Descriptor `sd`, der sich wie ein File Descriptor `fd` nutzen lässt

Alternative: `O_NONBLOCK`

- Dateien können auch mit Option `O_NONBLOCK` geöffnet werden, dann blockieren `read()`-Aufrufe nicht

```c
fd1 = open ("/tmp/1", O_RDONLY | O_NONBLOCK);
fd2 = open ("/tmp/2", O_RDONLY | O_NONBLOCK);
...
```

- Nachteil: Permanentes Polling mehrerer Dateien (ohne neue Daten) verbraucht unnötig Rechenzeit

Memory Mapped Files

- Der Inhalt einer Datei kann in den (virtuellen) Prozessspeicher eingeblendet werden
- Aufruf:

```
fd = open (filename, ...);
char *addr;
addr = mmap (NULL, len, prot, flags, fd, offset);
```

- `prot`:

```
PROT_EXEC:  Pages may be executed.
PROT_READ:  Pages may be read.
PROT_WRITE: Pages may be written.
PROT_NONE:  Pages may not be accessed.
```

- `flags`:

```
MAP_SHARED: Share this mapping. Updates to the
mapping are visible to other processes that map
this file [...]
MAP_PRIVATE: Create a private copy-on-write
mapping. [...]
```


Memory Mapped Files

- `mmap()` sucht freien Speicherbereich (im virtuellen Adressraum) für das Mapping
- Zugriff auf Speicheradressen in diesem Bereich wird zu Dateizugriff
- Häufig eingesetzt bei Datei, die aus gleich großen Datensätzen besteht:

  statt

  ```
  lseek (fd, i*recsize, ...); read (fd, &buf);
  ```

  Direktzugriff mit `f[i]`

Vorankündigung WS 2014/15

- Nächstes Semester: Vorlesung **Betriebssystem-Entwicklung mit Literate Programming**
- Informationen unter http://ohm.hgesser.de/be-ws2013/
  - 4 SWS (2+2, wie Systemprogrammierung)
  - Informatik / Medieninformatik: Schwerpunkt
  - Wirtschaftsinformatik: außerhalb des Schwerpunkts
  - Voraussetzungen: BS-Grundlagen, C-Kenntnisse, hilfreich: Systemprogrammierung